XMLRPC Support on VxWorks

Rich Neswold
<neswold@fnal.gov>

April 3, 2002

Abstract
A presentation on using the VxWorks implementation of the XMLRPC protocol.
Topics

- Brief review of the XMLRPC protocol
- Using the library on VxWorks
  - Using XmlRpc::Value classes
  - Using XmlRpc::Fault classes
  - Writing a client application
  - Writing a server application
- Live examples!
The XMLRPC Protocol

• Platform and language independant

• Uses HTTP as its transport

• Has several types of supported data
  – Integer, Double, Boolean, String, Date, Arrays, Structures, and Binary Data

• Parameters and results are described via XML documents

• Errors are reported through “faults”
A protocol example

A server has a function called `length()` which returns the number of characters contained in a string passed to it. The prototype for this interface would be:

```c
int length(string)
```

For this example, assume the client made the following request:

```c
length("Hello")
```
An Example (con’t)

The client converts the parameter to an XML document and loads the POST request with it. The document would look something like this:

```xml
<?xml version='1.0'?>
<methodCall>
    <methodName>length</methodName>
    <params>
        <param>
            <value><string>Hello</string></value>
        </param>
    </params>
</methodCall>
```
An Example (con’t)

The server will return an XML document representing the result:

```xml
<?xml version='1.0'?>
<methodResponse>
  <params>
    <param>
      <value><i4>5</i4></value>
    </param>
  </params>
</methodResponse>
```
Implementation

- Uses Duane Voy’s web server for VxWorks

- Written in C++
  - Uses RTTI and Exception Handling, so -frtti and -fexceptions compiler options are needed
  - Uses nested classes to simulate namespaces (will use namespace, when available)
The XmlRpc::Value Classes

- All are derived from XmlRpc::Value (which, itself, is abstract)
- XmlRpc::Value has no useful public methods
- Containers (arrays and structures) use pointers to XmlRpc::Value objects; use dynamic_casts to downcast
- Overloaded operators have been avoided
- Must use factory methods to create (prevents stack-based instances)
XmlRpc::Bool

Represents the XMLRPC boolean type.

- Bool* Bool::create(bool);
  Factory method which allocates a new Bool object.

- bool getValue();
  Returns the value of the object.
XmlRpc::Date

Represents the XMLRPC date type. NOTE: The XMLRPC protocol doesn’t specify whether the date is local or GMT; the communicating applications decide this.

• Date* Date::create(time_t = 0);
  Factory method which allocates a new Date object.

• time_t getValue();
  Returns the value of the object.
**XmlRpc::Double**

Represents the XMLRPC floating point type. The current specification only supports an optional sign character, followed by digits, optionally followed by a decimal point and digits – no scientific notation.

- `Double* Double::create(double);`
  
  Factory method which allocates a new Double object.

- `double getValue();`
  
  Returns the value of the object.
**XmlRpc::Integer**

Represents the XMLRPC 32-bit signed integer type.

- **Integer* Integer::create(int);**
  Factory method which allocates a new Integer object.

- **int getValue();**
  Returns the value of the object.
**XmlRpc::String**

Represents the XMLRPC string type.

- `String* String::create(string const&);`
  - `String* String::create(char const*);`

  Factory methods which allocates a new String object.

- `string const& getValue();`

  Returns the value of the object.
XmlRpc::Binary

Represents the XMLRPC binary type. Use this type as a last resort; structured data is much more desirable.

- Binary* Binary::create();
  Binary* Binary::create(uint8_t const*, size_t);

  Factory methods which allocates a new Binary object.

- BinData const& getValue();

  Returns the value of the object. A BinData object is a vector of 8-bit values. The data can be accessed by using the subscript operator.
XmlRpc::Struct

Represents the XMLRPC struct type.

- **Struct** Struct::create();
  Factory method which allocates a new Struct object.

- **void add(string const&, Value const*);**
  Adds a data type to the structure and associates it with a field name.

- **Value const* get(string const&);**
  Returns the data associated with the field name or NULL if it isn’t found.
XmlRpc::Array, (XmlRpc::Params)

Represents the XMLRPC array type.

- Array* Array::create();
  Factory method which allocates a new Array object.

- void append(Value const*);
  Expands the array and adds a data type to the end.

- Value const* get(size_t);
  Returns the data at the specified index. If the index is out of range, NULL is returned.

- size_t size();
  Returns the number of (top-level) elements in the array.
Using `XmlRpc::Value` Objects

This example creates an array of ten random integers.

```cpp
Array* a = Array::create();

for (size_t ii = 0; ii < 10; ++ii)
    a->append(Integer::create(rand()));
```

- The `append()` takes an `XmlRpc::Value*`. The `*::create()` return pointers to objects derived from `XmlRpc::Value`, so they can be used as arguments.

- Once you give an allocated object to a container, the container becomes the owner – even if you later extract it.

- This example works when there is a lot of heap available.
This example prints the integers in an array.

```
Array* a = ...; // Created elsewhere

for (size_t ii = 0; ii < a->size(); ++ii) {
    Integer const* v = dynamic_cast<Integer const*>(a->get(ii));
    if (v)
        printf("a[%u] = %d\n", ii, v->getValue());
    else
        printf("a[%u] isn’t an integer!\n", ii);
}
```
Error Handling

- XMLRPC handlers can only return one value (which may be an array or structure)
- The signify errors, a handler returns a fault
- Faults are essentially structures with two fields: faultCode and faultString
- OO languages generally map faults into their native exception handling mechanism.
The `XmlRpc::Fault` Classes

- All exceptions thrown by this module have `XmlRpc::Fault` as their base class.

- Due to design decisions, `Fault` pointers are thrown. This means the catcher is responsible for freeing up the memory.

- The XMLRPC specification doesn’t reserve any values for the error codes.
XmlRpc::Fault

The most general class used to report XMLRPC faults. This is also the base class for other fault classes.

- Fault(int, string const&);
  Fault(int, char const*);

  These constructors create a new Fault object.

- int getCode();

  Returns the error code of the fault.

- string const& getMessage();

  Returns the error message of the fault.
XmlRpc::MemFault

Indicates a memory problem caused the failure.

- MemFault();
  This constructor creates a new MemFault object. The error code is set to 800.
xmlrpc::ParseFault

Indicates the XML parser found a syntax error.

- ParseFault(char const*);

  This constructor creates a new ParseFault object. The error code
  is set to 801.
XmlRpc::ArgFault

This gets thrown when an XMLRPC handler doesn’t like the arguments passed to it.

- ArgFault(char const*);

  This constructor creates a new ArgFault object. The error code is set to 802. The string passed to the constructor will get sent to the caller (across the network.)
Using `XmlRpc::Value` Objects (revisited)

Let's redo our first example. This time, we'll make it more robust by handling possible faults.

```cpp
Array* a = Array::create();

try {
    for (size_t ii = 0; ii < 10; ++ii) {
        Integer const* v = Integer::create(rand());

        try { a->append(v); }
        catch (...) { delete v; throw; }
    }
}

catch (...) { delete a; throw; }
```
 STATUS getReading(char const* str)  
{  
    Server server("due12.fnal.gov", 4352, "/xmlrpc/Accelerator");  
    Request req("getReading");

    try {  
        req.addParam(String::create(str));

        Reply const* const rep = server.send(req);  
        Struct const* const s = dynamic_cast< Struct const* >(rep->result());

        printf("Device: %s\nValue: %f %s\n", str,  
               dynamic_cast< Double const* >((s->get("scaled")))->getValue(),  
               dynamic_cast< String const* >((s->get("units")))->getValue());
        delete rep;
    }
    catch (Fault* e) {  
        printf("Fault %d : %s\n", e->getCode(), e->getMessage());
    }
    return OK;
}
Building a Server Handler

To build a server-side handler for XMLRPC requests, the following steps must be taken:

- Write the handler
- Register the handler, along with any other related handlers, with a service
- Load, onto VxWorks, your new service’s object file after the XMLRPC module
Step-by-step Example

For this example, we’ll define an XMLRPC service named “Sample”. One of the functions in this service is hello(), which takes no arguments and returns the string “Hello, World!”.
Step: Write the Handler

```cpp
static Reply const* helloWorld(Request const& req) throw (Fault*)
{
    if (req.nArgs() == 0) {
        Reply* reply = new Reply;
        if (reply) {
            reply->addParam(String::create("Hello, World!"));
            return reply;
        } else
            throw new MemFault();
    } else
        throw new ArgFault("no parameters, please");
}
```
Step: Register with the Service

First we make sure our service is defined:

```cpp
static Service modSample("Sample",
    "This is a sample module. It contains several functions used "
    "to test the implementation and to test out client code.");
```

Next we register our handler:

```cpp
static MethodInfo const hdlr1(modSample,
    helloWorld, "hello", "string hello()",
    "An extremely boring procedure call. Simply returns "Hello, World!"");
```
**Step: Loading onto VxWorks**

Add commands to your VxWorks start-up script to load the module (assume, in this example, our module is called `sample.out`):

```c
ld < vxworks_boot/fe/deadoak/libmewstest2400.out
MEwsNew(80, 5, 100, "vxworks_boot/fe/deadoak/mews/", 0)
MEwsAddPrivileges(3, 0x83e18867, 0xffffffff)
ld(1,1,"vxworks_boot/module/PPC750/xmlrpc-latest.out")
ld(1,1,"sample.out")
```
** Time out for some real examples **
## Timing

<table>
<thead>
<tr>
<th>Function</th>
<th>Handler Time</th>
<th>Python</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>hello</td>
<td>1.0 mS</td>
<td>20.3 mS</td>
<td>38.9 mS</td>
</tr>
<tr>
<td>getTasks</td>
<td>3.5 mS</td>
<td>135.6 mS</td>
<td>(*) 62.5 mS</td>
</tr>
</tbody>
</table>

**Notes:**

- Measurements were made with `tcpdump`.
- Python times were from secondary calls (the communications were set up with the server during the first call.) The C++ times include this initialization.
- The Python times increase rapidly as the returned document gets more complicated. Alternate parsers will improve the performance.

(*) When sending the return value of `getTasks()` to `echo()`, the PowerPC took 750µS to parse the parameter and return a value.
Final Comments

Cons:

• Web server security needs to be vastly improved
  – Too much functionality in web server
  – Install IPFilter

• Parser is “open-ended” – it should probably restrict the size of the requests

• Network resources need to be improved in kernel
  – More socket handles
  – Support HTTP 1.1
  – Tune the garbage collector parameters
Final Comments

Pros:

- Easy to create handlers
- Can talk to clients from many operating systems and many, many programming languages
- Uses a standardized, published protocol
- Front-end is self-documenting